CS 103 Unit 8b Slides

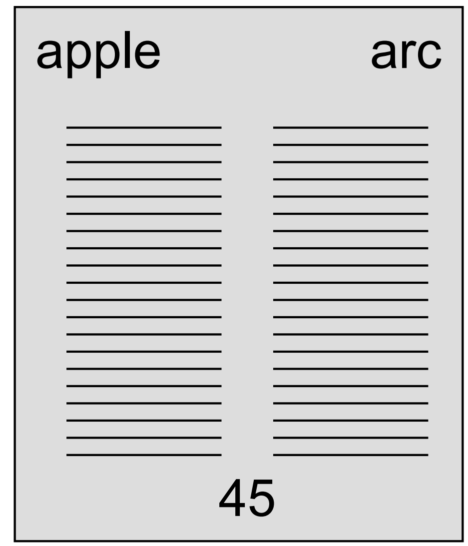
Algorithms

Mark Redekopp

ALGORITHMS

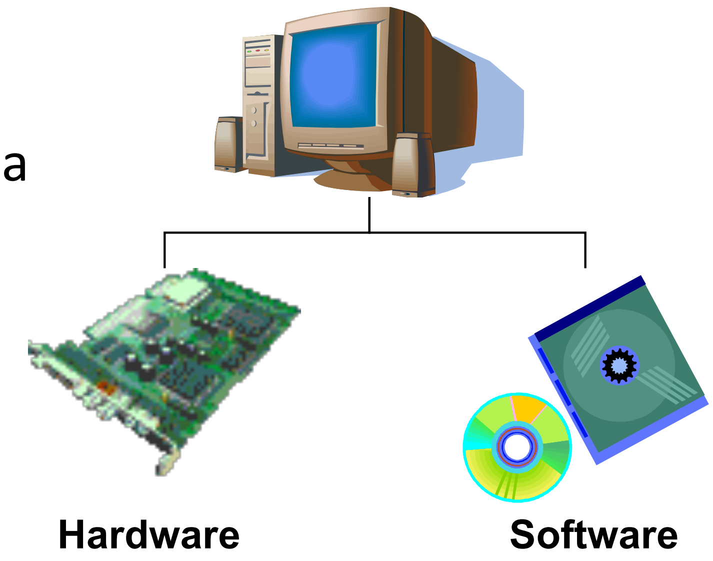
How Do You Find a Word in a Dictionary

- Describe an "efficient" method
- Assumptions / Guidelines
  - Let *target_word* = word to lookup
  - N pages in the dictionary
  - Each page has the *start* and *last* word on that page listed at the top of the page
  - Assume the user understands how to perform alphabetical ("lexicographic") comparison (e.g. "abc" is smaller than "acb" or "abcd")



Algorithms

- Algorithms are at the heart of computer systems, both in HW and SW
  - They are fundamental to Computer Science and Computer Engineering
- Informal definition
  - An algorithm is a precise way to accomplish a task or solve a problem
- Software programs are collections of algorithms to perform desired tasks
- Hardware components also implement algorithms from simple to complex



Humans and Computers

- Humans understand algorithms differently than computers
- Humans easily tolerate ambiguity and abstract concepts using context to help.
  - "Add a pinch of salt." How much is a pinch?
- Computers only execute well-defined instructions (no ambiguity) and operate on digital information which is definite and discrete (everything is exact and not "close to")

Formal Definition

- For a computer, “algorithm” is defined as...
  - ...an ordered set of unambiguous, executable steps that defines a terminating process
- Explanation:
  - **Ordered Steps**: the steps of an algorithm have a particular order, not just any order
  - **Unambiguous**: each step is completely clear as to what is to be done
  - **Executable**: Each step can actually be performed
  - **Terminating Process**: Algorithm will stop, eventually. (sometimes this requirement is relaxed)

Algorithm Representation

- An algorithm is not a program or programming language
- Just as a story may be represented as a book, movie, or spoken by a story-teller, an algorithm may be represented in many ways
  - Flow chart
  - Pseudocode (English-like syntax using primitives that most programming languages would have)
  - A specific program implementation in a given programming language

Algorithm Example 1

- List/print all factors of a natural number, **n**
  - How would you check if a number is a factor of **n**?
  - What is the range of possible factors?

```
i ← 1
while(i <= n) do
  if (remainder of n/i is zero) then
    List i as a factor of n
  i ← i+1
```

F T

- An improvement

```
i ← 1
while(i <= sqrt(n) ) do
  if (remainder of n/i is zero) then
    List i and n/i as a factor of n
  i ← i+1
```

Algorithm Time Complexity

- We often judge algorithms by how long they take to run for a given input size
- Algorithms often have different run-times based on the input size [e.g. # of elements in a list to search or sort]
  - Different input patterns can lead to best and worst case times
  - Average-case times can be helpful, but we usually use worst case times for comparison purposes

Big-O Notation

- Given an input to an algorithm of size n, we can derive an expression in terms of n for its worst case run time (i.e. the number of steps it must perform to complete)
- From the expression we look for the dominant term and say that is the big-O (worst-case or upper-bound) run-time
  - If an algorithm with input size of n runs in $n^2 + 10n + 1000$ steps, we say that it runs in $O(n^2)$ because if n is large n^2 will dominate the other terms

Pseudocode	Steps
i ← 1	1
while(i <= n) **do**	1*n
if (remainder of n/i is zero) **then**	2*n
List i as a factor of n	1*n
i ← i+1	1*n

5n+1
= O(n)

Big-O Notation

- Given an input to an algorithm of size n, we can derive an expression in terms of n for its worst case run time (i.e. the number of steps it must perform to complete)
- From the expression we look for the dominant term and say that is the big-O (worst-case or upper-bound) run-time
  - If an algorithm with input size of n runs in $n^2 + 10n + 1000$ steps, we say that it runs in $O(n^2)$ because if n is large n^2 will dominate the other terms
- Main sources of run-time: Loops
  - Even worse: Loops within loops (i.e. execute all of loop 2 w/in a single iteration of loop 1, and repeat for all iterations of loop 1, etc.)

i ← 1	**1**
while(i <= n) **do**	**1*n**
if (remainder of n/i is zero) **then**	**2*n**
List i as a factor of n	**1*n**
i ← i+1	**1*n**

5n+1
= O(n)

Algorithm Example 1

- List/print all factors of a natural number, **n**
  - What is a factor?
  - What is the range of possible factors?

  i ← 1
  while(i <= n) **do**
   if (remainder of n/i is zero) **then**
    List i as a factor of n
   i ← i+1

O(n)

- An improvement

  i ← 1
  while(i <= sqrt(n)) **do**
   if (remainder of n/i is zero) **then**
    List i and n/i as a factor of n
   i ← i+1

O($\sqrt{n}$)

Algorithm Example 2a

- Searching an ordered list (array) for a specific value, k, and return its index or -1 if it is not in the list
- Sequential Search
  - Start at first item, check if it is equal to k, repeat for second, third, fourth item, etc.

myList	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

i ← 0

while (i < length(myList)) **do**

  if (myList[i] equal to k) **then return** i

  else i ← i+1

return -1

Algorithm Example 2b

- Sequential search does not take advantage of the ordered nature of the list
  - Would work the same (equally well) on an ordered or unordered list
- Binary Search
  - Take advantage of ordered list by comparing k with middle element and based on the result, rule out all numbers greater or smaller, repeat with middle element of remaining list, etc.

k = 6

List	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

Start in middle

6 < 9

List	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

6 > 4

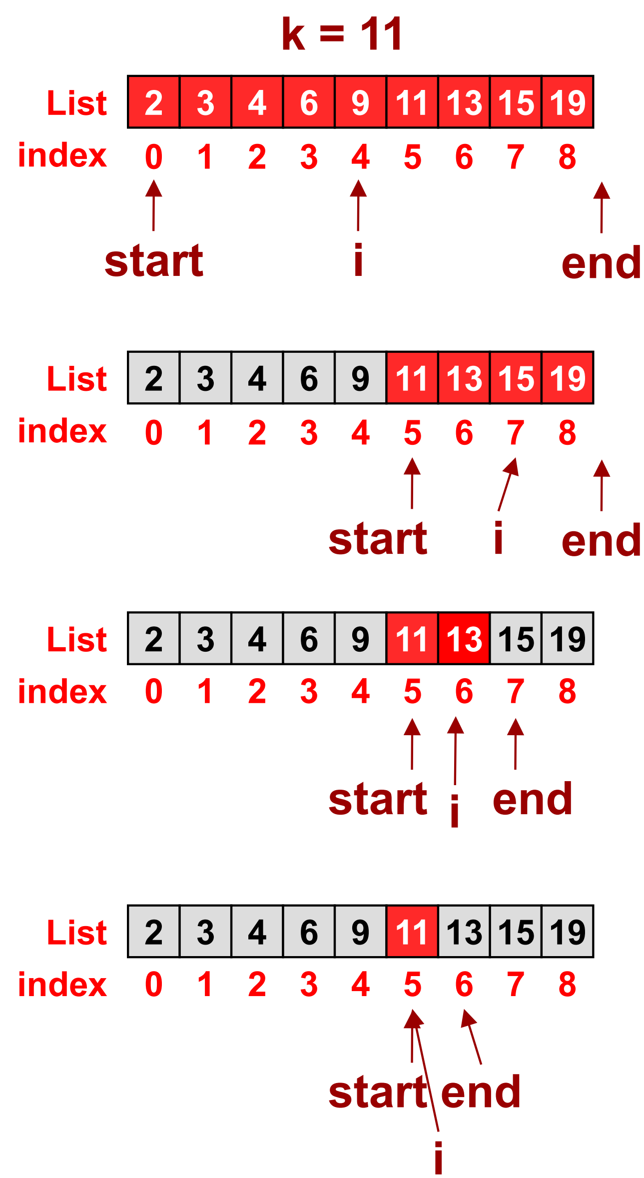
List	2	3	4	6	9	10	13	15	19
index	0	1	2	3	4	5	6	7	8

6 = 6

Algorithm Example 2b

- Binary Search
  - Compare k with middle element of list and if not equal, rule out ½ of the list and repeat on the other half
  - Implementation:
    - Define range of searchable elements = [start, end)
    - (i.e. start is inclusive, end is exclusive)

```
start ← 0; end ← length(List);
while (start index not equal to end index) do
  i ← (start + end) /2;
  if ( k == List[i] ) then return i;
  else if ( k > List[i] ) then start ← i+1;
  else end ← i;
return -1;
```



Sorting

- If we have an unordered list, sequential search becomes our only choice
- If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
- Many sorting algorithms of differing complexity (i.e. faster or slower)
- Bubble Sort (simple though not terribly efficient)
  - On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e. w/o the newly placed maximum value)

List	7	3	8	6	5	1
index	0	1	2	3	4	5

Original

List	3	7	6	5	1	8
index	0	1	2	3	4	5

After Pass 1

List	3	6	5	1	7	8
index	0	1	2	3	4	5

After Pass 2

List	3	5	1	6	7	8
index	0	1	2	3	4	5

After Pass 3

List	3	1	5	6	7	8
index	0	1	2	3	4	5

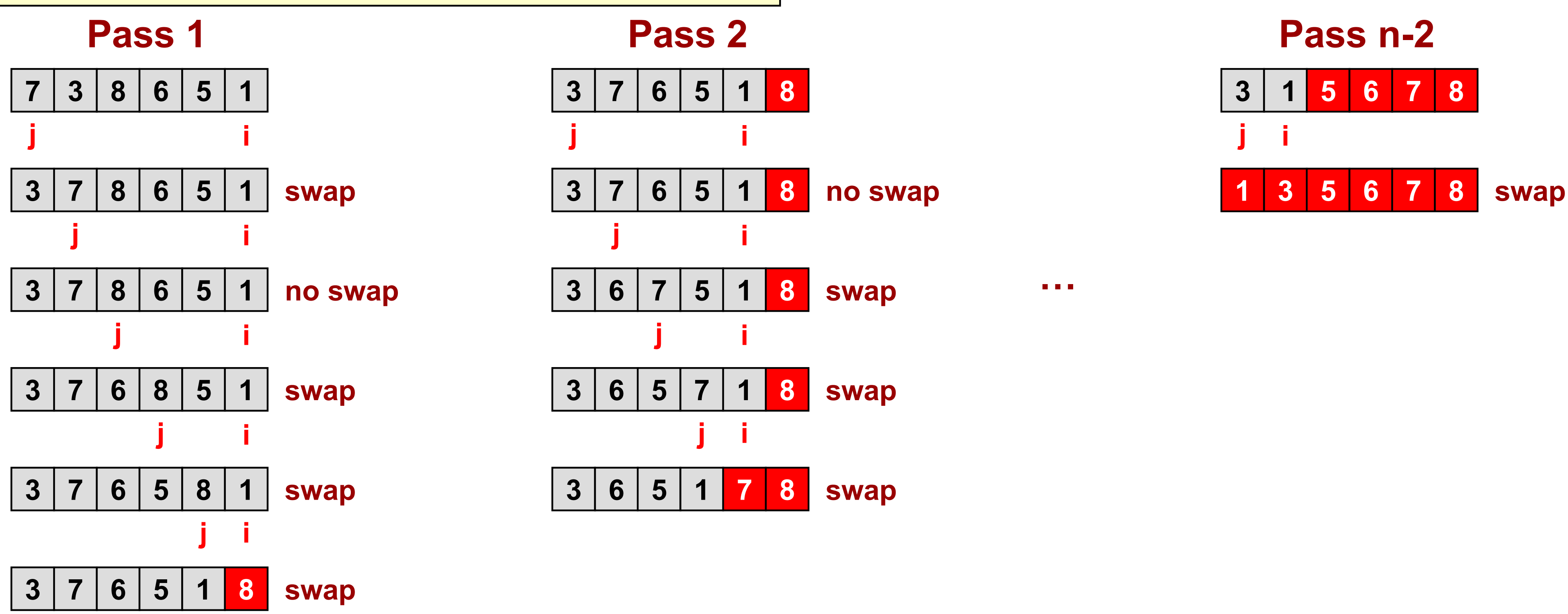
After Pass 4

List	1	3	5	6	7	8
index	0	1	2	3	4	5

After Pass 5

Bubble Sort Algorithm

```
void bsort(int* mylist, int n)
{
  int i ;
  for(i=n-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(j, j+1)
  }  }  }
}
```

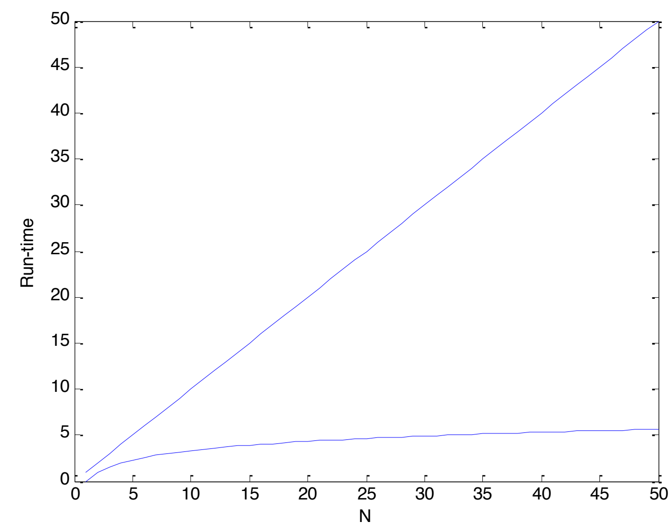


Complexity of Search Algorithms

- Sequential Search: List of length n
  - Worst case: Search through entire list
  - Time complexity = an + k
    - a is some constant for number of operations we perform in the loop as we iterate
    - k is some constant representing startup/finish work (outside the loop)
  - Sequential Search = O(n)
- Binary Search: List of length n
  - Worst case: Continually divide list in two until we reach sublist of size 1
  - Time = $a*\log_2 n + k = O(\log_2 n)$
- As n gets large, binary search is far more efficient than sequential search

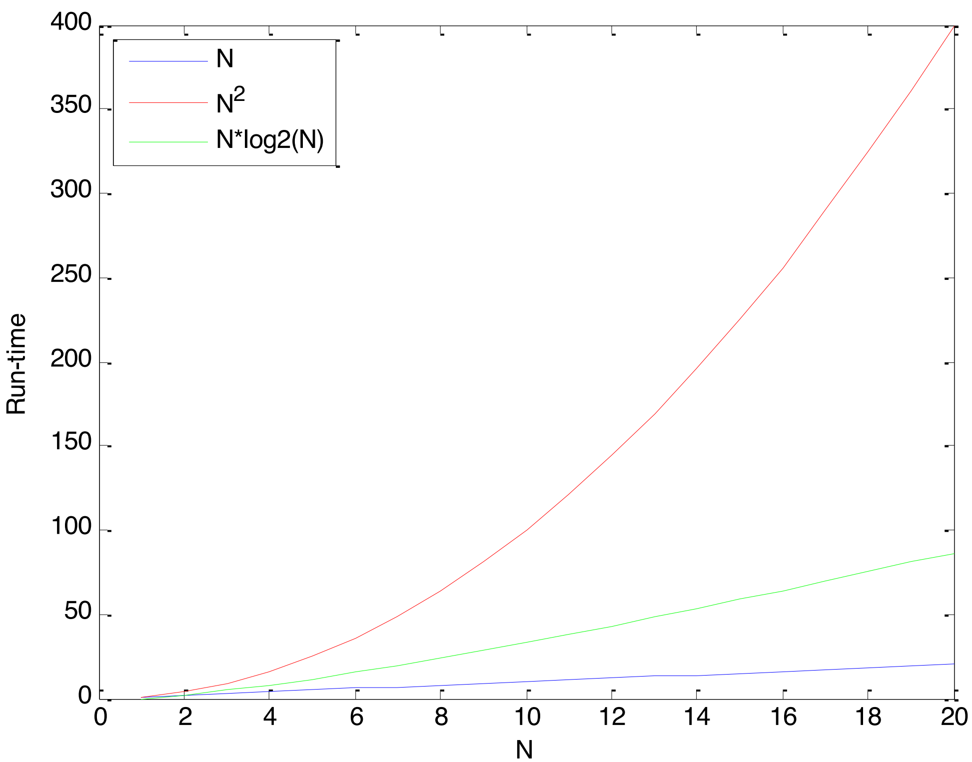
Multiplying by 2 k-times yields: $2*2*2\ldots*2 = 2^k$

Dividing by 2 k-times yields: $n / 2^k = 1$
$k = \log_2 n$



Complexity of Sort Algorithms

- Bubble Sort
  - 2 Nested Loops
  - Execute outer loop n-1 times
  - For each outer loop iteration, inner loop runs i times.
  - Time complexity is proportional to:
    n-1 + n-2 + n-3 + … + 1 =
    $(n^2 + n)/2 = O(n^2)$
- Other sort algorithms can run in $O(n*\log_2 n)$



Importance of Time Complexity

- It makes the difference between effective and impossible
- Many important problems currently can only be solved with exponential run-time algorithms (e.g. $O(2^n)$ time)…we call these NP = Non-deterministic polynomial time algorithms) [No known polynomial-time algorithm exists]
- Usually algorithms are only practical if they run in P = polynomial time (e.g. $O(n)$ or $O(n^2)$ etc.)
- One of the most pressing open problems in CS: "Is NP = P?"
  - Do P algorithms exist for the problems that we currently only have an NP solution for?

N	$O(1)$	$O(\log_2 n)$	$O(n)$	$O(n*\log_2 n)$	$O(n^2)$	$O(2^n)$
2	1	1	2	2	4	4
20	1	4.3	20	86.4	400	1,048,576
200	1	7.6	200	1,528.8	40,000	1.60694E+60
2000	1	11.0	2000	21,931.6	4,000,000	#NUM!